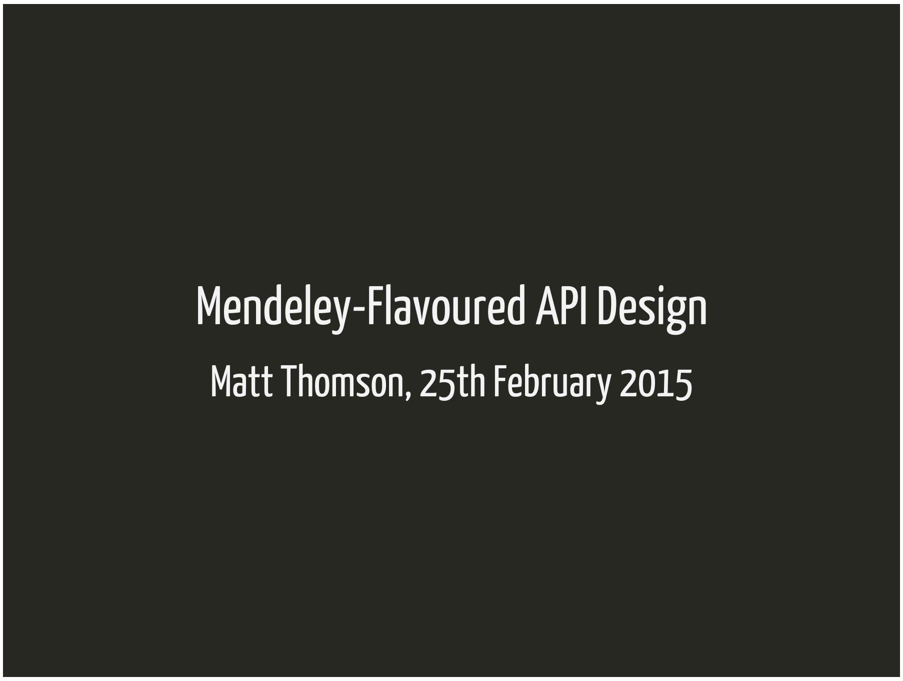


Mendeley-Flavoured API Design

Matt Thomson, 25th February 2015

What's this all about?

- Transferring knowledge about API design
- Some general principles, then lots of specifics
- Some facts, lots of opinions
- Chance to ask lots of questions
- Biscuits

- Knowledge gathered over last nine months of API work
- Lots to talk about, we've got two hours
- Hope not to need full time, won't be upset if people leave early
- Much of what follows will be an opinion - more than way to do it, this is what we think and why

Opinions

✔ We do this, and I like it.

✖ We don't do this, but that's fine because I don't like it anyway.

⚠ We do this, but I wish we didn't.

? This is something that we haven't done that's worth at least thinking about. It may still be a terrible idea.

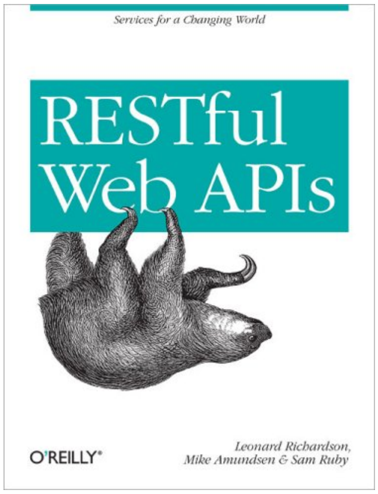
Rough agenda

- General advice
- Guiding principles
- REST
- HTTP specifics
  - URLs
  - Methods
  - Status codes
  - Headers
- Common patterns
- Compatibility and versioning
- Gnarly bits
- Race conditions

- Lots of topics
- Key points are at the start, later bits dive into specifics

General advice

Read this book


Services for a Changing World
RESTful Web APIs
O'REILLY
Leonard Richardson, Mike Amundsen & Sam Ruby

- We don't agree with everything in it, but it's still worth a read.
- PDF floating around
- Old version on my desk (more XMLy - use the new one)

Use other APIs for inspiration (but don't copy blindly)

- Most of the problems we try to solve are not unique to us
- We like Twitter, Stripe, Twilio, ...
- Just because someone else is doing it doesn't mean it's right, or even that the people who put it there are happy with it
- Right now someone in Twitter is giving a talk on things they wish they'd done differently in their API
- Other people's guides:
  - https://github.com/interagent/http-api-design
  - https://github.com/gocardless/http-api-design
  - https://github.com/WhiteHouse/api-standards

A tidy API is more important than tidy code

- Design the API that your clients want - there are more of them than us.
- Might make implementation more complicated but usually worth it
- Exception: database constraints (hard to change schemas on our big tables, had to make compromises)
- Won't say much about implementation in this talk

Look before you leap

- APIs can be hard to change after they're released
- We can move very quickly, but clients can't always
- Some decisions are hard to change

Guiding principles

What are the specific constraints that apply to our API?

Support a wide range of clients

- Network
  - Same Amazon region as the API
  - Data centre with a fibre link
  - Home internet connection
  - Mobile network
- Device
  - Servers
  - PCs
  - Mobile phones / tablets
- Programming languages
- Syncing vs non-syncing
- Release cycles

- multiple devices: Differ vastly in power, battery.
- languages: not all languages have the same support for concurrency/HTTP features as Java/Dropwizard - sometimes more, sometimes less.
- syncing/non: Getting lots of data vs on-demand
- release cycles: can vary widely between webapps (days) and installed apps (months when you consider how long it takes people to upgrade)

Be consistent

Should look like one API, not many.

Similar things should look similar, regardless of which service they're in, or who worked on them.

See https://wiki.mendeley.com/display/JAR/API+conventions for conventions to follow.

Examples:

- prefer snake_case to camelCase or kebab-case
- a person has a first_name and a last_name
- when to use "link", "website" and "url"

- antipattern: if you can work out team structure by looking at the API
- can be difficult - microservices, big team

Include all non-trivial business logic

Having logic spread amongst all of our clients makes it much harder to:

- achieve cross-device consistency
- write new apps

- Applies even if you're building for one client - prepare for a second
- Clients should be little more than a view on the data. That data, and the bulk of the logic, should be in the API

Follow standards pragmatically

Use standard HTTP, and industry best practices, where appropriate:

- avoids surprising client developers
- tried and tested
- better support in client-side libraries

However, there may be times where:

- following the standard exactly would be excessively complicated
- there is no standard
- there are multiple competing standards, and vigorous debate about which one is right

---

- More important to have a working API with actual clients, than achieve perfect RESTful purity

REST

- This section is a bit abstract. Will try and tie it down to specifics.

What is REST?

> Representational State Transfer (REST) is a software architecture style consisting of **guidelines and best practices** for creating scalable web services. REST is a coordinated **set of constraints** applied to the design of components in a distributed hypermedia system that can lead to a **more performant and maintainable architecture**.
>
> REST has gained widespread acceptance across the Web *(citation needed)* as a simpler alternative to SOAP and WSDL-based Web services. RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) used by web browsers to retrieve web pages and send data to remote servers.

(Source: Wikipedia)

- Based on Roy Fielding's PhD thesis (2000)
- I haven't read this, but probably should have, and you probably should.

REST constraints

Client-server

a uniform interface separates clients from servers.

Stateless

no client context is stored on the server between requests.

Cacheable

clients can cache responses.

Layered system

a client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.

- Client-server: clients don't have to worry about data storage (improves portability), well-defined interface means that servers and clients can evolve independently
- Stateless: improves scalability, clients can have local state
- Cacheable: responses can define themselves as cacheable or not, improves performance
- Layered system: load balancing, caching, security
- optional one called "code-on-demand" - servers changing client behaviour by sending code to clients (e.g. Java applets). Don't propose we do this.

Uniform interface

Identification of resources

- each resource is identified by a URI, each request relates to a resource
- separation between resource and representation (JSON/XML)

Manipulation of resources through representations

- a client that holds a representation has enough information to modify/delete the resource

Self-descriptive messages

- message holds enough information about how to process it

Hypermedia as the engine of application state (HATEOAS)

- clients make state transition using links in the messages
- warning - here be dragons

- Uniform interface decouples the architecture, allows client and server to evolve independently
- Resources are the entities in the system, representations are the message contents
- Messages describe the state of the resources, and how they should change
- REST usually runs over HTTP, but it doesn't have to. I'll assume you know about HTTP, and talk about how we use it to design APIs.

What does this mean in practice?

- Interfaces need to be clearly defined, to allow clients and servers to evolve independently
- URLs need to be chosen carefully
- Use appropriate HTTP methods to manipulate state
- Use standard HTTP headers to indicate how messages should be processed
  - e.g. Content-Type header indicates which parser to use
  - e.g. Cache-Control header indicates how long to cache
- Think in terms of maniuplating resources, rather than making remote procedure calls
  - e.g. change the "is read" flag on a document, rather than marking a document as read
- Think about making resources cacheable

Hypermedia

API design strategy for connecting resources within APIs

- each response includes links to related/nested resources, e.g.
  - group has link to list of group members
  - document has link to attach a file
- clients don't hard-code URLs, they start at the root and navigate to the thing that they want

---

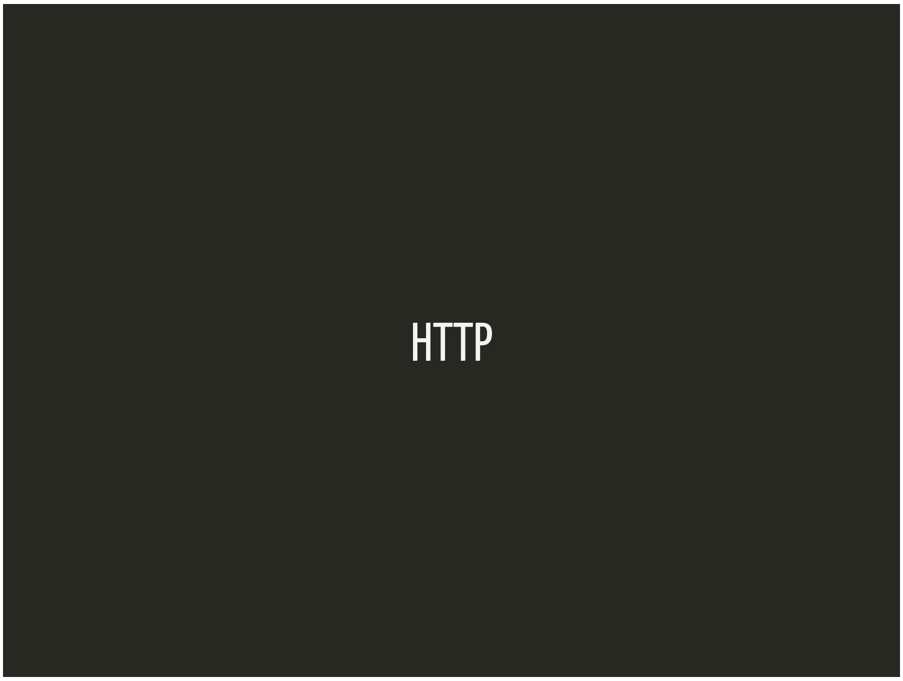
- navigating hypermedia is like browsing a website
- if you want to build a useful client, instead of hard-coding URLs, you end up hard-coding which links to follow
- this is just my view, most API people disagree, so do read up on this and make up your own mind
- book on Joyce's desk which I didn't understand

Look out for flame wars

> …
>
> Right, yes but in doing so your knuckles become that much closer to the ignorant ground. Clearly you haven't read Section 6.3.3.1 in which Fielding clearly states that the HTTP protocol, as it stands, is not enough to support high-performance concurrent connections and, in fact, proposes two new protocols: MGET and MHEAD which deal more directly with MIME response which is a much nicer implementation then –
>
> Right, YES BUT YOU should know that MGET and MHEAD were REJECTED because they VIOLATED many RESTful conditions – specifically that before a RESTful request could be made, the requestor would have to make all of its requests at ONCE because it wouldn't actually know the length of the request prior to sending and requests are bound by length and it's fundamentally –
>
> …

(Source: Someone save us from REST)

- This is a parody (happened when Rails switch from PUT to PATCH for updates), but is actually fairly accurate
- No real REST standards beyond Roy's PhD thesis => lots of interpretations
- api-craft email list: worth joining, but watch out for simple API questions turning into big hypermedia debates



HTTP

- We're going to talk a bit about HTTP
- I'll assume you're fairly familiar with it, and drill into specific bits that are relevant to APIs.

Use the right spec

> Don't use RFC2616. Delete it from your hard drives, bookmarks, and burn (or responsibly recycle) any copies that are printed out.

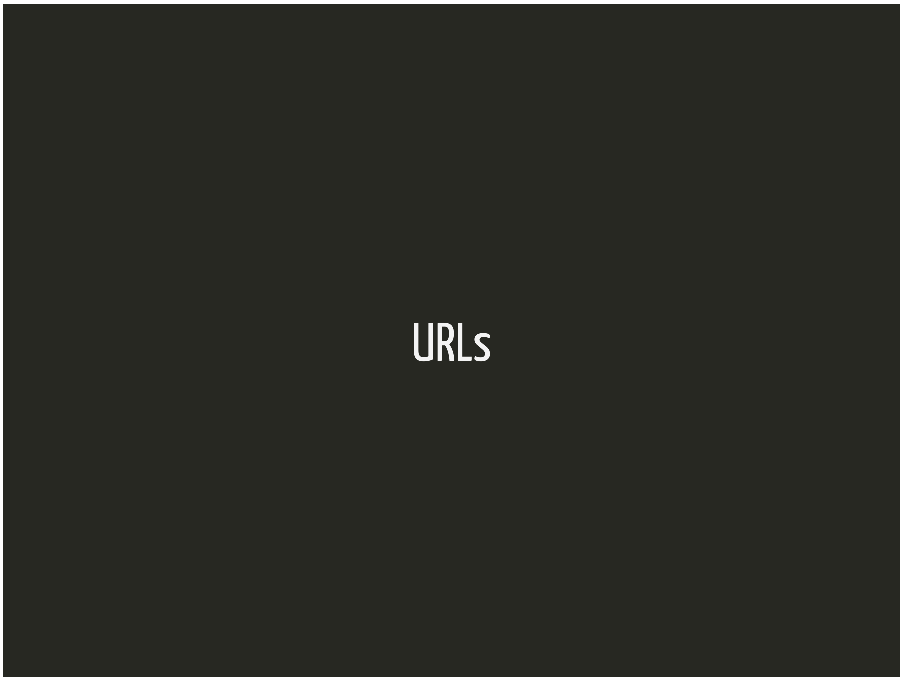
Source: mnot's blog

Reworked into these six RFCs:

- RFC7230 - HTTP/1.1: Message Syntax and Routing - low-level message parsing and connection management
- RFC7231 - HTTP/1.1: Semantics and Content - methods, status codes and headers
- RFC7232 - HTTP/1.1: Conditional Requests - e.g., If-Modified-Since
- RFC7233 - HTTP/1.1: Range Requests - getting partial content
- RFC7234 - HTTP/1.1: Caching - browser and intermediary caches
- RFC7235 - HTTP/1.1: Authentication - a framework for HTTP authentication

---

- RFC2616 was the reference for HTTP, but now it's deprecated. You'll still see it referred to in lots of places.



URLs

Basics

URLs are based around plural nouns.

- `/things` - identifies a collection of resources.
- `/things/(uuid)` - identifies a single resource within a collection.

- Should only need one UUID per URL - "universally unique"
- UUIDs rather than MySQL keys - don't couple the API to our database

Filter with query parameters

✔ `/documents?starred=true`

✔ `/files?document_id=b91d9530-9017-4b96-b000-cdc245920355`

✖ `/documents/b91d9530-9017-4b96-b000-cdc245920355/files/`

- reduces surface of the API
- makes it possible to get all files without having to traverse documents (better for syncing clients)

Prefer flat to nested

✓ `/files/0c98cfaa-7d58-4720-abd1-3b09fe008c48`

✗ `/documents/b91d9530-9017-4b96-b000-cdc245920355/files/0c98cfaa-7d58-4720-abd1-3b09fe008c48`

- reduces coupling between resources

If you have to nest

⚠ `/folders/1016198b-f97a-4123-82af-be7239ff352b/documents/8d7a4bd2-918a-4a8c-8529-4ca437b91313`

✖ ✖ `/oapi/library/documents/7135271181/file/3221525ea6f746b577b6a8ad40d89df3f41f776a/2589311`

- provide context by putting something between the IDs
- remove as many extraneous parts of the URL as you can
- think again about whether you really have to nest - can you identify the resource by one UUID?

- Second URL - IDs are document, filehash, group
- Not clear what that last ID is
- The resource in question is a file - why can't it have an ID and be identified by just that?

Actions

Some actions don't naturally fit a RESTful model.

As an emergency manoeuvre, use a POST with a verb in the URL.

⚠ `POST /trash/8d7a4bd2-918a-4a8c-8529-4ca437b91313/restore`

? `POST /trash/8d7a4bd2-918a-4a8c-8529-4ca437b91313/actions/restore`

✖ `RESTORE /trash/8d7a4bd2-918a-4a8c-8529-4ca437b91313`

This should be a last resort - prefer to PATCH the resource if you can.

- this is the only time you're allowed to put a verb in the URL
- prefer "/actions" - if you have multiple then "restore" is like an ID, so should have something in the middle
- however tempting it may be, don't invent your own HTTP method - some clients won't be able to handle this

Looking up by a secondary ID

⚠ `/catalog?doi=10.1016%2Fj.molcel.2009.09.013`

- `/catalog` is a collection resource, and needs to return a list (e.g. `/catalog?author=John+Smith`).
- But, with a DOI, this can only a list of 0 or 1 documents.
- Would be more natural to have something that could return a 200/404 status.

? `/catalog;doi=10.1016%2Fj.molcel.2009.09.013`

Matrix URIs

- Does exactly the thing we want
- Based on a 1996 blog post by Tim Berners-Lee - not in any RFC
- Not widely implemented
  - Jersey does support it (`@MatrixParam`), but most other frameworks don't

HTTP Methods

The big five

GET

Fetch a representation of the resource at the URI.

POST

Store the enclosed entity as a subresource of the resource at the URI.

PUT

Store the enclosed entity under the URI.

DELETE

Delete the resource at the URI.

PATCH

Apply partial modifications to the resource at the URI.

PATCH wasn't in the original specification, but we use it.

Safety and idempotency

A method is **safe** if it has no side effects.

- GET

A method is **idempotent** if the side effects of repeating the request are the same as for a single request.

- GET
- PUT
- DELETE

Otherwise, no guarantees about what repeating a request will do.

- POST
- PATCH (but often idempotent in practice)

Clients can use these facts to decide whether to retry a request.

- Safe method: retrying has same response
- Idempotent method: retrying may have different response (e.g. already deleted), but doesn't make any material difference
- Other methods won't be retried

Which one to use?

- Use GET for retrieving data.
- Use POST for creating a new resource (not idempotent - multiple POSTs create multiple resources).
- For updates:
    - PUT - replace the whole resource with this representation.
    - PATCH - replace only the fields that are in the body.

✓ We prefer PATCH to PUT for updates.

- Reduce window conditions caused by concurrent updates (more on this later).
- Don't need to have the whole resource in hand to do an update.
- Can still get PUT-like semantics by sending the whole object.
- In reality, the client doesn't replace the whole resource anyway (e.g. last modified time).

Specials

OPTIONS

- Returns details about what requests will be accepted for a URL.
- ✓ Used for CORS (more later).

HEAD

- Identical to GET, but returns only the headers, not the body.
- ? Clients could use this to get pagination counts, but we haven't implemented that.

Oddities

CONNECT

- ✖ Converts the connection to a transparent TCP/IP tunnel.

TRACE

- Echoes the request back, so that the client can see what intermediate servers have done.
- ✖ Breaks layered system constraint.

LINK / UNLINK

- Used to create/delete a link between two resources.
- e.g. adding/removing documents from folders.
- **?** Specified by this internet draft, but didn't make it into the HTTP spec.

MISCELLANEOUSOTHERTHING

- ✖ You can make up your own methods, but please don't.

CORS (cross-origin resource sharing)

- Ordinarily, web browsers block JavaScript from making HTTP requests to different domains, for security reasons.
- CORS is a way that browsers can negotiate with an API about whether they're allowed to make a request.

```
OPTIONS /documents
Origin: app.example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Authorization

----------

200 OK
Access-Control-Allow-Origin: app.example.com
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers: Authorization
```

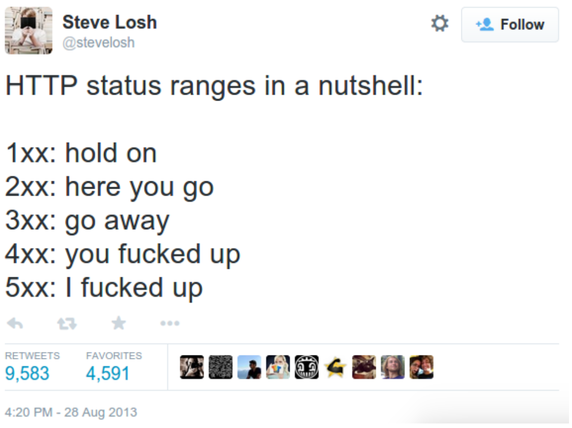
✓ CORS is enabled across our API, and is used in the web library.

More info (Wikipedia)

- Some APIs make you opt in to CORS, or specify the domain you want to use - we weren't aware of any reason to restrict it.

Status Codes

Status codes


Steve Losh
@stevelosh

HTTP status ranges in a nutshell:

1xx: hold on
2xx: here you go
3xx: go away
4xx: you fucked up
5xx: I fucked up

RETWEETS 9,583 FAVORITES 4,591

4:20 PM - 28 Aug 2013

✓ Use the right status code.

- SIP has 6xx, which is "everyone fucked up".
- Well-behaved client might want to retry on a 5xx (limited number of times, backoff), but shouldn't on a 4xx (definition of insanity)
- Beyond that, use the most appropriate code. Not going to give a list here (many in RFC 7231, many spread around other specs Wikipedia has a good list, as does appendix A of the book), but they all have very specific meanings, so read them and pick the most appropriate one.

Headers

- Method is the verb, URL identifies a resource, body contains a representation, so headers are the only other place to indicate how a message should be processed - use them
- Not going to go into lots of detail here - just mention a few patterns

Media types

✓ Resources should have a vendor-specific media type.

```
Content-Type: application/vnd.mendeley-document.1+json
```

- The "1" is a version number.
- Media type represents a JSON format on the wire, not the class in your implementation.
    - A GET and a PATCH should share a media type, even if the PATCH has fewer fields.
- Try to re-use media types between services.

✓ It's possible to route requests to different services based on the media type.

- Will almost always be JSON. Only exception I'm aware of is BibTeX.

Links

✓ Use Link headers to indicate when one resource is linked to another.

✓ Link headers are allowed (and useful) on requests as well as responses. For example:

```
POST /files
Link: </documents/291d3064-4f74-4932-bfc8-4277d441705b>; rel="document";
```

Custom headers

✓ Use standard headers wherever you can. Appendix B of the RESTful Web APIs book contains a handy list.

If you want to introduce a custom header:

- read appendix B of the RESTful Web APIs book.
- think really hard about whether you still need it.
- read appendix B of the RESTful Web APIs book again.

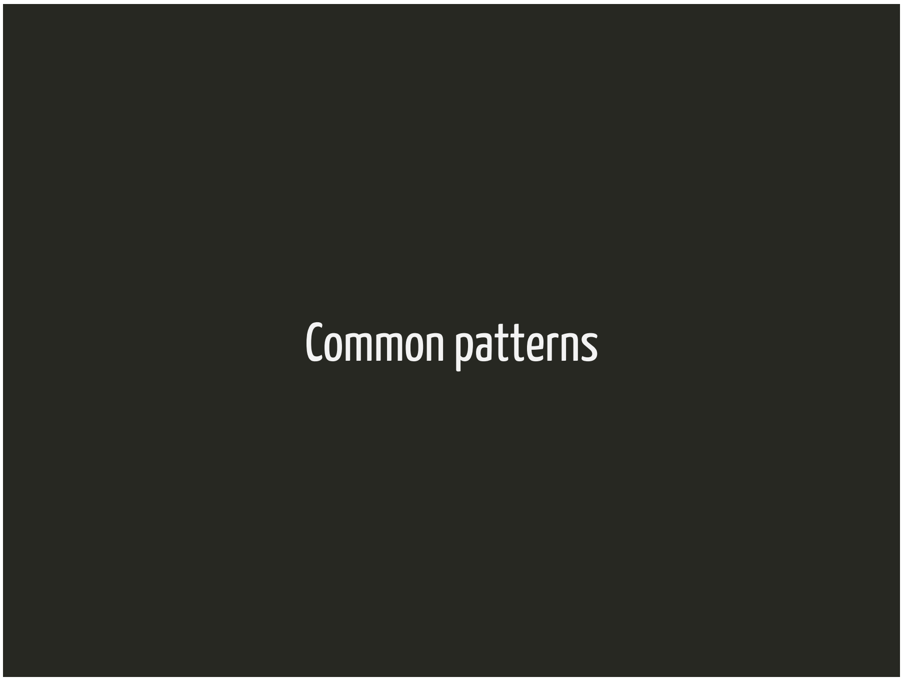
If you still want to introduce a custom header:

- give it a name starting with `Mendeley-`.

✓ `Mendeley-Count`

✗ `X-Mendeley-TraceID`

- Headers starting with "X-" used to be a best practice, but now considered deprecated - RFC 6648.



Common patterns

Creating resources

✓ The response to a POST should be 201 Created, with a Location header containing the URL where the resource can be found.

⚠ The body should contain a representation of the resources (including any server-generated fields).

```
POST /things

{"field": "value"}

----------

201 Created
Location: https://api.mendeley.com/things/291d3064-4f74-4932-bfc8-4277d441705b

{"field": "value", "created": "2015-02-18T04:57:56Z"}
```

This means that all clients have to download the (potentially very large) representation, even if they don't care about the server-generated fields.

? Use the HTTP Prefer header:

- `Prefer: return=minimal`
- `Prefer: return=representation`

- We didn't know about Prefer header at the time - wish we had. Gives clients a way to indicate whether they want to get a full representation or not.

Symmetry

The body of a POST request, and the body returned on a GET request to the URL from the Location header, should have the same structure.

```
POST /things

{"field": "value"}

----------

201 Created
Location: https://api.mendeley.com/things/291d3064-4f74-4932-bfc8-4277d441705b
```

```
GET /things/291d3064-4f74-4932-bfc8-4277d441705b

----------

200 OK

{"field": "value", "created": "2015-02-18T04:57:56Z"}
```

- GET response can have more fields, but that's OK - same structure is the important thing.
- e.g.
  - POST /files: body is the file bytes
  - GET /files/(id): body is the file bytes
  - extra metadata (e.g. the document that the file is attached to) goes in the headers as it's the only place left

Views

Problem

- Some objects (e.g. documents) are very large
- Some clients (e.g. mobiles) don't want to download the full content
- We don't want to do expensive database queries for data that the client doesn't need

Solution

⚠ Offer a choice of views.

- `GET /documents/291d3064-4f74-4932-bfc8-4277d441705b` - returns a minimal set of fields
- `GET /documents/291d3064-4f74-4932-bfc8-4277d441705b?view=bib` - returns extra fields for bibliographies
- `GET /documents/291d3064-4f74-4932-bfc8-4277d441705b?view=patent` - returns extra fields for patents
- `GET /documents/291d3064-4f74-4932-bfc8-4277d441705b?view=all` - returns the entire Montgomery

- I don't know how well this has worked in practice. Certainly internal clients generally just get the "all" view
- Some APIs (e.g. Facebook) go one step further and let you pick exactly which fields you want

Pagination

All APIs that return a collection must have an upper bound on the number of items in the response.

✓ Use **cursor-based** pagination. Return links to other pages (first, next, last, previous) in Link headers.

```
GET /documents

----------
200 OK
Link: </documents?marker=291d3064-4f74-4932-bfc8-4277d441705b>; rel="next";

[
  // documents
]
```

All of this is handled by the mendeley-pagination library.

- Keeps body as a representation.
- Works well with many HTTP clients (Jersey, Requests)
- Confuses some third parties who don't look at the docs (HOW VERY DARE THEY)

Bulk requests

✖ We occasionally get asked for an API like: `GET /profiles/id1,id2,id3`, to return a list of profiles by ID.

- different response formats from the same URL (`GET /profiles/id` shouldn't return a list)
- breaks cacheability
- URL doesn't represent a resource
- URLs have a maximum length (can get about 55 UUIDs in)

Bulk requests

Usually we get these requests because we have an API somewhere that returns a list of IDs, that should return a list of objects.

In this case, our followers API was returning profile IDs - it should return profiles.

✖

```
GET /followers
----------
200 OK

[
    {
        "profile_id": "00a4fb19-2c8f-472f-8f7f-72aa4dbf068b"
    },
    {
        "profile_id": "231994e9-f179-4b89-a7a9-be3cadf2da4e"
    }
]
```

- Made-up example, the real one is slightly different to this, but same principle.

Bulk requests

Usually we get these requests because we have an API somewhere that returns a list of IDs, that should return a list of objects.

In this case, our followers API was returning profile IDs - it should return profiles.

✓

```
GET /followers
----------
200 OK

[
    {
        "id": "00a4fb19-2c8f-472f-8f7f-72aa4dbf068b",
        "first_name": "Tommy",
        "last_name": "Cannon"
    },
    {
        "id": "231994e9-f179-4b89-a7a9-be3cadf2da4e",
        "first_name": "Bobby",
        "last_name": "Ball"
    }
]
```

- Detective work to find out what clients really want
- Our implementation might just get all of the individual items, but it's easier for us to do that in parallel than for some of our clients

Synchronization

Problem

- Syncing clients (e.g. desktop) want to have all of the user's data cached locally.
- Inefficient to download everything on every sync for very large libraries.

Solution

- Offer `modified_since` and `deleted_since` parameters on GET requests.
- On each sync, the client calls the API with `modified_since` set to the time that the previous sync started, then does the same for `deleted_since`.
- This gives enough information to bring the local database up-to-date.

- Large libraries: >10k documents

Synchronization - alternative

? Provide an API that returns a list of changes:

```
GET /deltas/documents?since=(timestamp)

200 OK
[
  {
    "type": "modified",
    "document": {
      "id": "00000000-0000-0001-0000-000000000002",
      "title": "Underwater basket weaving"
    }
  },
  {
    "type": "trashed",
    "document": {
      "id": "00000000-0000-0003-0000-000000000004",
      "title": "Overground bucket sawing"
    }
  },
  {
    "type": "deleted",
    "document": {
      "id": "00000000-0000-0005-0000-000000000006"
    }
  }
]
```

Validation errors

✖ We've been returning these as 400 Bad Request:

```
400 Bad Request

{"message": "title may not be null"}
```

✔ Starting to use a more specific status code, and a more structured response:

```
422 Unprocessable Entity

[{"field": "title", "message": "may not be null"}]
```

- Status provides context, body provides detail
- Should errors be suitable for end-users?
    - My view - possibly, but at the very least there should be enough information between the message and the status code to come up with a message.

Compatibility and versioning

The golden rule

Once we've released an API, we must not make breaking changes to it

(even if it's not being used, or if it's only deployed to staging, or if there's only one client, or if we marked it as beta, or if all of the clients are sitting in the same room as us, or if we're really-really-careful-yes-we-really-mean-it-this-time)

- Client teams have felt in the past that we change APIs too often and that they're trying to hit a moving target.
- Our culture of continuous deployment doesn't help
- Need to build trust that we're a stable platform
- Advertising instability isn't enough - people might not read the docs, they'll still be annoyed if it changes, and if someone builds a cool app on a beta API it would be a bad idea to break it.

What changes are OK?

✓ Adding new API resources.

✓ Adding new optional request parameters to existing API methods.

✓ Adding new properties to existing API responses.

✓ Changing the order of properties in existing API responses.

? Changing the length or format of object IDs or other opaque strings.

✓ Fixing a 5xx server error.

- Aiming to give clearer guidance about what changes we'll allow ourselves to make - we can't expect clients to upgrade on every change, so need to agree guidelines so that we can make changes safely
- List originated from Stripe
- This is the theory - we only put this list together recently, so changing some of these things might break clients
- I'm not sure we can safely change our object IDs - if they stopped being UUIDs then all hell would break loose
- Last one is my own
- Kevin's hackday project

What changes are bad?

✖ Renaming or removing API resources.

✖ Adding new required request parameters to existing API methods.

✖ Renaming, moving or removing properties from existing API responses.

✖ Changing the structure of an API response (e.g. list -> object).

✖ Changing the status code of an API response (in general).

- There might be some cases where it's OK to replace one error with a different one. Needs to be thought about.

How can I make breaking changes?

Before we've publicly released an API:

- Announce plans to affected client teams, and get their agreement.
- Run old and new in parallel for a week.
- Switch off the old.

- This is what we've done - clients much preferred it to us changing things under their feet
- Do this even if you've only got one client and they can move quickly - better to play by our own rules and build trust
- Communication is key
- This approach might be suitable for things we've released externally and marked as beta - but still need to notify everyone (not just people who've hit it - we won't build trust unless we're seen to be following our own rules).

How can I make breaking changes?

After we've publicly released an API:

- We haven't done this yet.

Some thoughts:

? Offer the new version under a different content type (`application/vnd.mendeley-document.2+json`).

✖ Put new versions under different URLs.

✔ Blackout tests.

? Consider client release cycles, and monitor usage.

- Same principle (run old and new together) but need to allow much more time
- We've put version numbers in our content types, but yet to see whether it'll work in practice.
- I'm concerned we're putting too much significance on "version 2" - some other APIs version with dates so that they're less attached to them. Would be a shame if we couldn't make breaking changes before a significant release.
- Don't change URLs - the resources themselves haven't changed, just their representations
- Communication, communication, communication

Gnarly bits

Trash

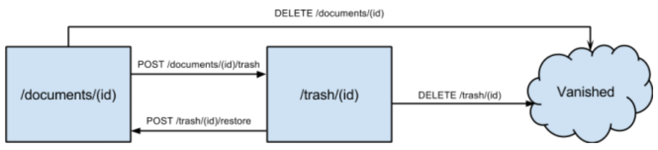
⚠ I hate trash.

Goal

Many clients don't care about the trash, and they shouldn't need to.

Split documents into /documents and /trash URLs

State transitions



- Split also allows us to show, at the API level, that trashed documents can't be modified (no PATCH /trash/(id))

Trash

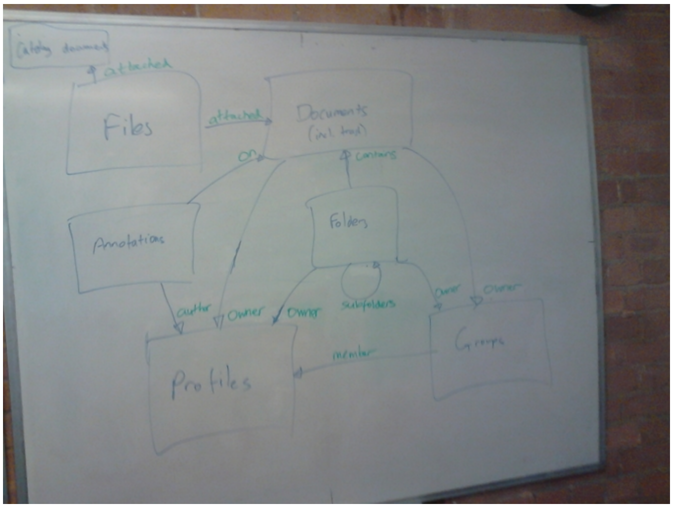
Problems with this:

- A syncing client that doesn't care about trash still needs to call `GET /trash?modified_since=...`, to find out what things have been moved to the trash.
- Every group has its own trash, but this isn't reflected in the desktop UI.
- Trash leaks out into other resources - should `GET /files` include files attached to trashed documents?
    - We decided it shouldn't, unless you add `?include_trashed=true`.

⚠ I hate trash.

"Every group has its own trash" - was a surprise to us, caused issues with web library

Everything is connected



- Diagram shows core API resources - documents, files, folders, annotations, profiles, groups
- Arrows show where one references another (e.g. a document is in a group)
- Something to watch when changing things (changes can be wide-reaching)
- Try not to couple new stuff in so closely

Dates

✓ Use ISO 8601 format for dates and times: `2015-02-18T04:57:56Z`

⚠ Unfortunately, some standard HTTP headers use their own (really ugly) format, defined in RFC 2822: `Thu, 01 May 2014 10:07:28 GMT`

This is annoying for clients, but can't be helped.

✓ Use server-generated dates everywhere - the server is the only reliable clock.

- Annoying but not too hard - all of the languages I tried can convert dates in one or two lines
- Could use RFC 2822 everywhere, but ugly and locale-specific

Side effects

Beware of cases where `PATCH /resource1` can affect the state of `/resource2`.

Try to decouple into two operations, so that the client has to be explicit about updating both - but often this is not possible.

✖ I don't have a good answer for this.

Client-specific behaviour

There are too many places in the API where different clients get different behaviour.

Things we've done:

✖ added extra "client data" fields, for specific clients to store their own attributes

```
{
  "title": "Underwater basket weaving",
  "client_data":"{\"desktop_id\":\"1bcd0e7c-5846-4c22-bed8-cc3d5685ef4a\"}"
}
```

✖ restricted certain endpoints to particular client IDs (lots of whitelists spread through the code)

✖ kept some endpoints quiet in the hope that no-one else would find them

---

- Too many "whitelists"
- What if we had to change the desktop client ID? How many code changes would we have to deploy to keep it working, and do we even know what they all are?

Client-specific behaviour

What to do about this?

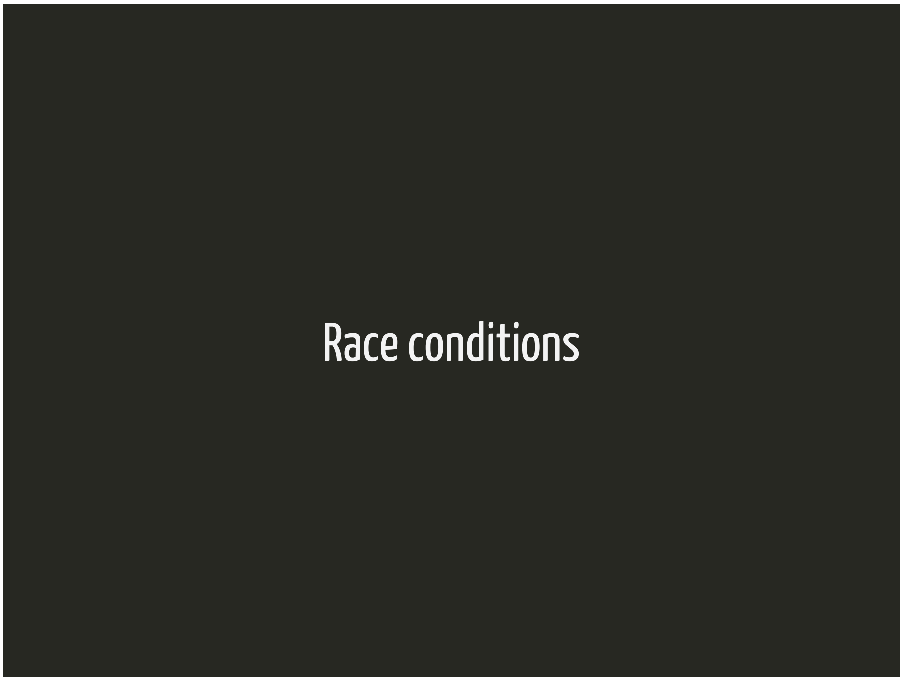
- Ideally, nothing - if we're willing to put it into a client, it should go in the API.
- Exceptions to the above:
    - if you're building a platform-internal service that doesn't form part of the client API
    - if there are lawyers at your desk and they want to have a meeting

? If we really have to lock down a service down, consider using OAuth scopes.

- clients request the specific extra privileges that they need
- OAuth database says which clients are allowed which scopes - avoids having whitelists spread through lots of services

---

- platform-only
    - allows reuse along microservice boundaries
    - restrict based on client ID



Race conditions

- No locking/synchronization on an API
- State can change between calls
- Here are some tactics that can help

Concurrent updates

Remember the semantics of PUT and PATCH:

- PUT - replace the whole resource with this representation.
- PATCH - replace only the fields that are in the body.

Suppose that two clients want to update different fields on the same resource.

- PUT - both send in the full object, and the second one will overwrite the first one's changes.
- PATCH - both send in only the field they want to update, and both updates are applied correctly.

✔ If used like this, PATCH helps to reduce (but not avoid) race conditions.

✖ However, most of our clients use PATCH like PUT (sending the whole representation every time), so they don't get this benefit.

Concurrent updates

✓ Clients that want PUT-like semantics, but want to avoid lost updates, can specify the timestamp of the previous time they retrieved a resource.

```
PATCH /documents/291d3064-4f74-4932-bfc8-4277d441705b
If-Unmodified-Since: Thu, 01 May 2014 10:07:28 GMT

{"title": "Underwater basket weaving"}

----------
412 Precondition Failed
```

(see also: ETags).

? Could make this required, and send back `428 Precondition Required` if there is no If-Unmodified-Since header.

- Only implemented for documents - bigger objects, so more likely to change under your feet - but this was just due to time constraints, so might be worth rolling out across the board
- Can also be used for GET (don't download data that hasn't changed since the client last requested it)

Preventing duplicate creates

Suppose a client doesn't get a response to a POST. Should it retry?

- can't retry automatically (POST is not idempotent)

? Consider using the post once exactly pattern.

```
POST /poe/documents
----------
200 OK
Location: /poe/documents/291d3064-4f74-4932-bfc8-4277d441705b
```

```
POST /poe/documents/291d3064-4f74-4932-bfc8-4277d441705b

{"title": "Underwater basket weaving"}
----------
200 OK
Location: /documents/7ab2c167-8e48-4fb8-85b0-73cdb8662a64
```

```
POST /poe/documents/291d3064-4f74-4932-bfc8-4277d441705b

{"title": "Underwater basket weaving"}
----------
405 Method Not Allowed
Allow: GET
```

- our clients just retry (should be fairly rare, worse case is a duplicate document created)
- only a draft (not an RFC), but not updated in 10 years
- draft is slightly different (more about HTML forms) but idea is the same

Questions?

- in the pub?